# Chap 5
# Stimulus and Response


教育部顧問室
「超大型積體電路與系統設計」教育改進計畫
EDA聯盟編製

# Overview

- Simple Stimulus
- Verifying the Output
- Self-Checking Testbenches
- Complex Stimulus
- Complex Response
- Predicting the Output
- Summary



## Simple Stimulus

- Verifying the Output
- Self-Checking Testbenches
- Complex Stimulus
- Complex Response
- Predicting the Output
- Summary



# Generating a Simple Waveform

```
reg clk;
parameter  cycle=10;
always
begin
     #(cycle/2);
      clk=1'b0;
     #(cycle/2);
    clk=1'b1;
end
```

* 50%  duty-cycle clock



# Generating Synchronized Waveform

```
always
begin
    #50 clk=1'b0;
    #50 clk=1'b1;
end
initial
begin
    rst=1'b0;
    #150  rst=1'b1;
    #200  rst=1'b0;
end
```

* there is a race condition between clk and rst signals
* how to solve it?



# Solve the Race Condition

```
always
begin
    #50 clk <= 1'b0;
    #50 clk <= 1'b1;
end
initial
begin
    rst=1'b0;
    #150  rst <= 1'b1;
    #200  rst <= 1'b0;
end
```

* use the non-blocking assignment



# Non-Zero Delay Generation of Synchronous Data

```
initial
begin
   rst=1'b0;
   #50  clk<=1'b0;
   repeat (2)     #50  clk<= ~clk;
   rst<= #1  1'b1;
   repeat (4)     #50  clk<= ~clk;
   rst<=#1   1'b0;
end
```

//What if it were necessary to reset the device under verification multiple times during the execution of a testbench ?



# Encapsulating the Generation of a Synchronized Waveform

```
always
begin
    #50  clk <=1'b0;
    #50  clk <=1'b1;
end

task hw_reset;
begin
     rst =1'b0;
     wait (clk !==1'bx);
     @(negedge clk);
     rst<=1'b1;
    @(negedge clk);
    @(negedge clk);
     rst<=1'b0;
end
endtask;
initial hw_reset;
```



# Abstracting Waveform Generation

Using synchronous test vectors to verify a design is rather cumbersome .

→ hard to interpret and difficult to correctly specify.

**1 .Try to apply the worst possible combination of inputs .**

**2 .Pass input values as arguments to the subprogram .**

**3 .Stimulus generated with abstracted operations is easier to write and maintain .**



- Simple Stimulus
- Verifying the Output
  - Self-Checking Testbenches
  - Complex Stimulus
  - Complex Response
  - Predicting the Output
  - Summary



# Sampling Using the $monitor task

```
initial
begin
        $monitor("…",rst,d0,d1,sel,q,qb);
end
```

*change in values of signals rst,d0,d1,…cause the display of simulation results.



# Visual Inspection of Waveforms

However，waveform displays usually provide a more intuitive visual representation of simulation results

It is a tool-dependent process that is different for each language and each tool



- Simple Stimulus
- Verifying the Output
- Self-Checking Testbenches
- Complex Stimulus
- Complex Response
- Predicting the Output
- Summary



# Self-Checking Testbenches

A reliable and reproduceable technique for output verification ：testbench that verify themselves

We must automate the process of comparing the simulation results against the expected output



# Automating Output Verification

Step 1: include the expected output with the input stimulus for every clock cycle

Step2 : golden vectors(a set of reference simulation results)

⇒ If the simulation results are kept in ASCII files ,the simplest comparison process involes using UNIX *diff* utility.

--must still be visually inspected

--do not adapt to change

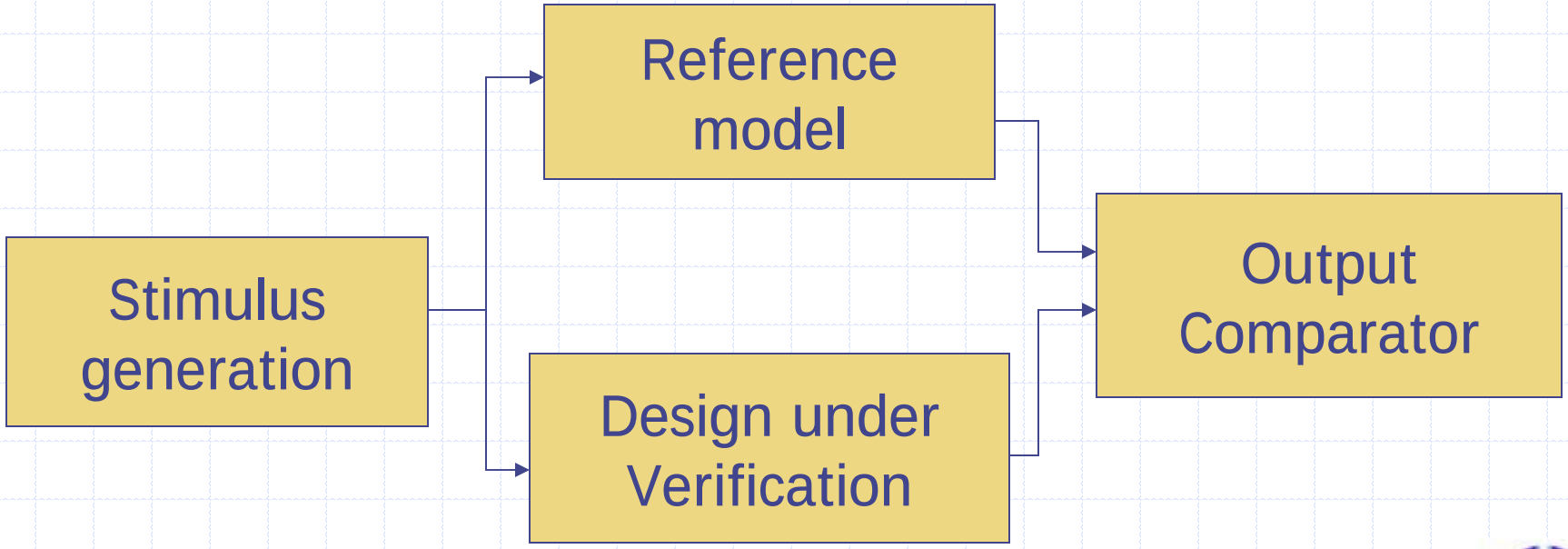
--require a significant maintenance effort



# Run-Time Result Verification

Using a reference model (a extension of golden vector)

however , in reality, a reference model rarely exist.



# Model the Expected Response

Include the verification of the operation's output as part of the subprogram .

Integrate both the stimulus and response checking into complete operations .



- Simple Stimulus
- Verifying the Output
- Self-Checking Testbenches
- Complex Stimulus
- Complex Response
- Predicting the Output
- Summary



# Complex Stimulus

More complex stimulus generation scenarios through the use of bus-functional models .

If the interface being driven contains handshaking or flow-control signals ,the generation of the stimulus requires cooperation with the design under verification .



# Feedback between stimulus and design

Without feedback , verification can be under constrained .

Wait for feedback before proceeding



# Wait for feedback

```
procedure bus_request is
      variable cycle_count : integer=0;
begin
   req<=‘1’;
   wait until clk=‘1’;
   while grt=‘0’  loop
        wait until  clk=‘1’;
        cycle _count := cycle_count +1;
   end loop  ;
   assert  1<=cycle_count and cycle_count<=5;
end   bus_request;
                              // a example of verifying the bus request operation
                              // deadlock !
```



# Wait for feedback & Avoid deadlock

```
procedure bus_request is
      variable cycle_count : integer=0;
begin
   req<=‘1’;
   wait until clk=‘1’;
   while grt=‘0’  loop
        wait until  clk=‘1’;
        cycle _count := cycle_count +1;
        assert  cycle_count<500
             report “Arbiter is not working”
             severity  failure;
   end loop ;
   assert  1<=cycle_count and cycle_count<=5;
end   bus_request;
```



# Asynchronous Interfaces

```
//Many interfaces , although implemented using FSM and
//  edge-trigered flip-flops , are specified in  asynchronous fashion .
task bus_request;
output good;
begin
        reg=1'b1;
        fork :wait_for_grt
            #60                  disable wait_for_grt;
            @(posedge  grt)    disable wait_for_grt;
        join
        good=(grt==1'b1);
end
endtask
                        //verifying the asynchronous bus request operation
```



- Simple Stimulus
- Verifying the Output
- Self-Checking Testbenches
- Complex Stimulus

Complex Response

- Predicting the Output
- Summary



# Complex Response

**Def** : something that cannot be verified in the same process that generate the stimulus .

=> definitely not verifiable using visual inspection of waveforms .

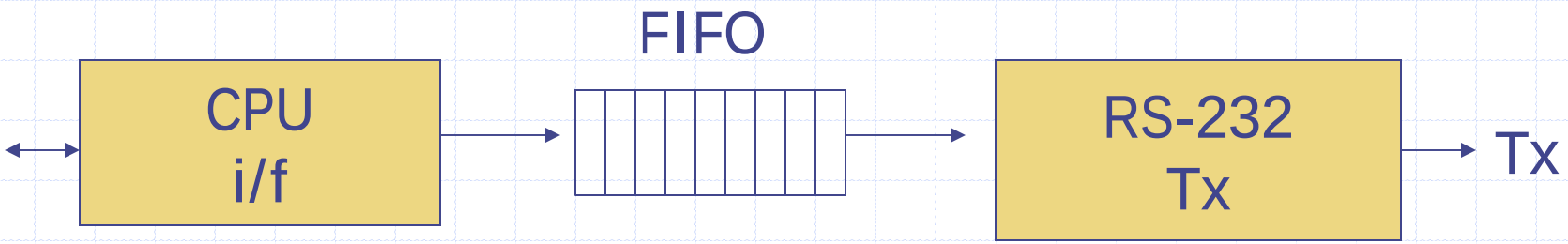
Latency and output protocols create complex responses .



# Complex Response

* Universal Asynchronous Receiver Transmitter (UART)

Because the RS-232 protocol is slow , waiting for the output corresponding to the last CPU write cycle would introduce huge gaps in the input stimulus.




A simple design can have a complex response .

# Handling Unknown or Variable Latency

**Stimulus and response could be implemented in different execution threads**

```
event sync;
initial
begin : stimulus
   …
   -> sync;
   …
end

initial
begin: response
   …
   @(sync);
   …
end
```



# Abstracting Output Operation

The output operations ,encapsulated using *tasks* in verilog, take as argument the value expected to be produced by the design.

The most flexible implementation for a output operation monitor is to simply return to the caller whatever output value was just received.

--Separate monitoring from value verification



# Monitoring Multiple Possible Operations

```
load  A,R0
load  B,R1
add   R0,R1,R2
sto   R2,X
load  C,R3
add   R0,R3,R4
sto   R4,Y
```

* many possible execution orders

How to write an encapsulated output monitor ?

**Write an operation “dispatcher” task or procedure .**



- Simple Stimulus
- Verifying the Output
- Self-Checking Testbenches
- Complex Stimulus
- Complex Response
- Predicting the Output
- Summary



# Predicting the Output

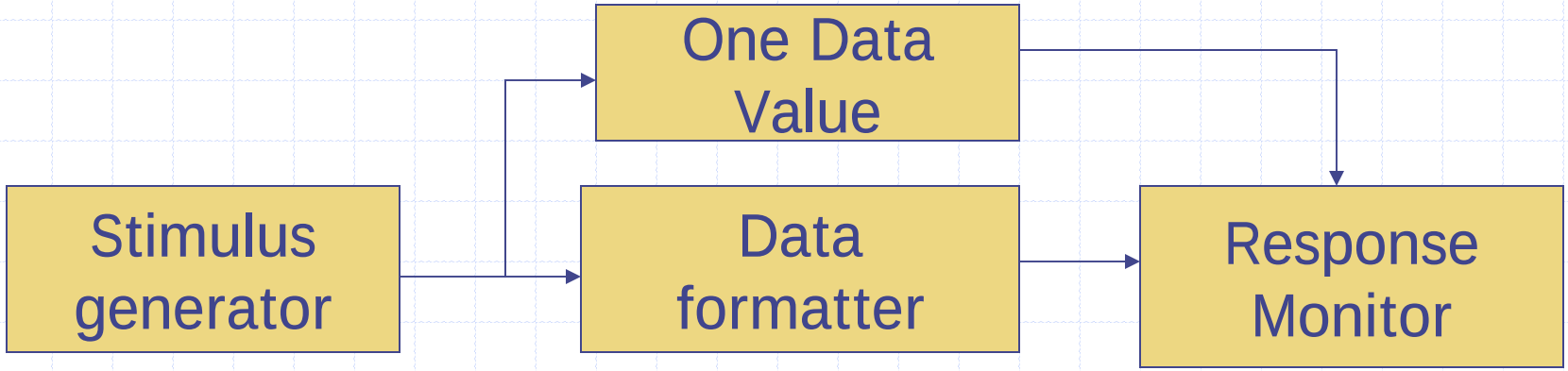
When implementing self – checking testbenches , we should have detailed knowledge of the output to be expected.

Knowing exactly which output to expect and how it can be verified to determine functional correctness is the most crucial step in verification .



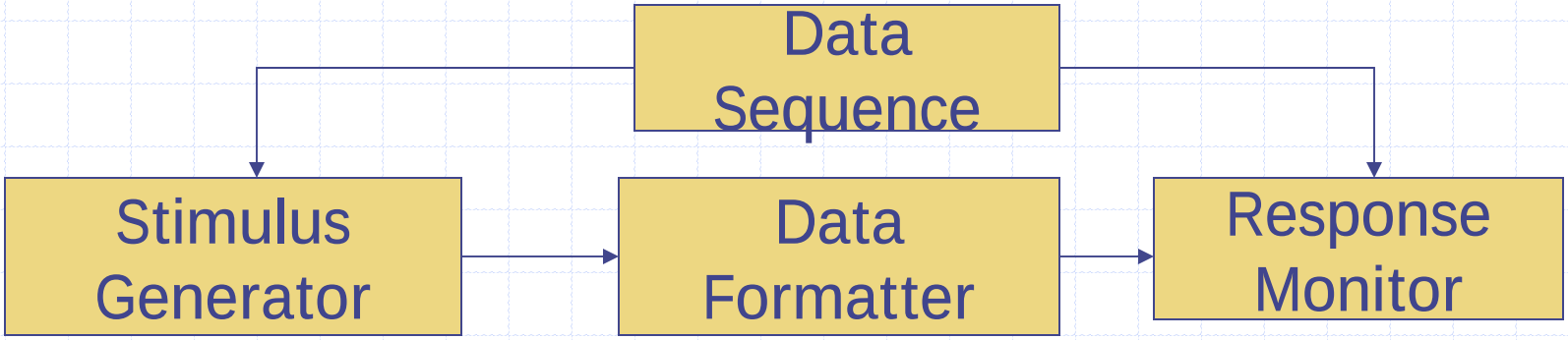
# Predicting the Output

There is a class of design where the input information is not transformed , but simply reformatted .



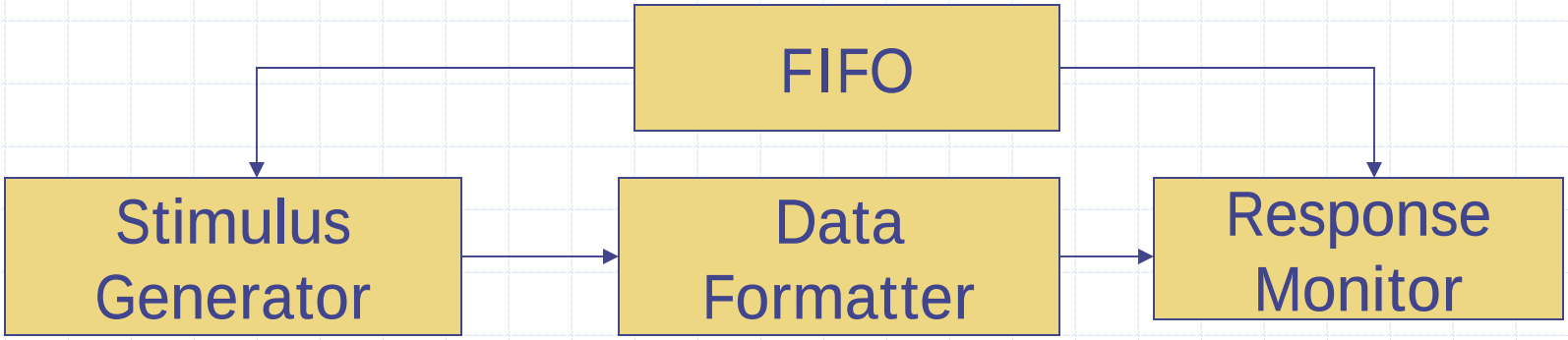
# Predicting the Output

If the input sequence is short and pre-determined , using a global data sequence table is the simplest approach.



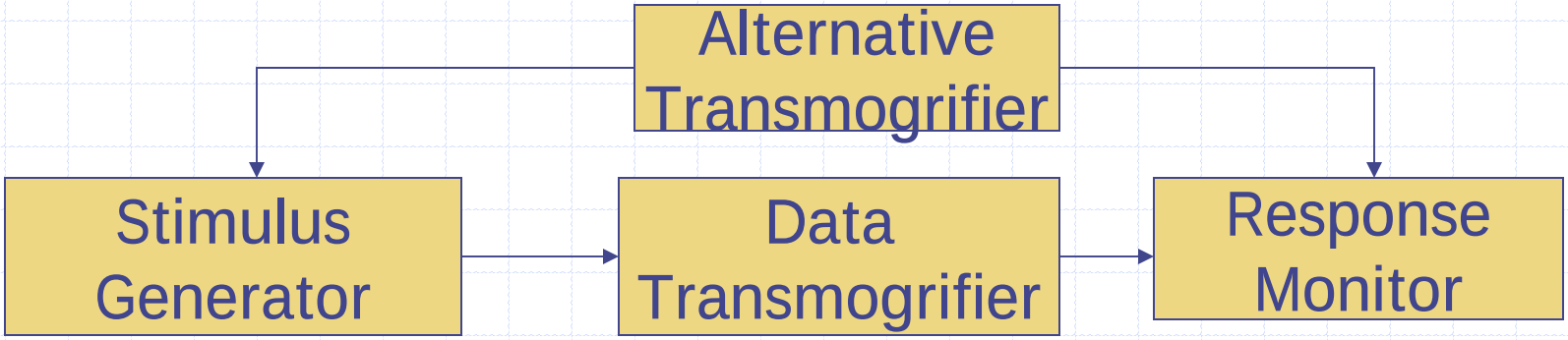
# Predicting the Output

Long data sequence can use a FIFO between the generator and monitor



# Predicting the Output

Some design processes and transforms the input data completely and thoroughly.



Simple Stimulus

Verifying the Output

Self-Checking Testbenches

Complex Stimulus

Complex Response

Predicting the Output

Summary



# Summary

**Using bus-functional models to generate stimulus and monitor response.**

*abstract the interface operations and remove the testcases from the detailed implementation of each physical interface.

**Make each individual testbench completely self-checking .**

*The expected response must be embedded in the test-bench at the same time as the stimulus.